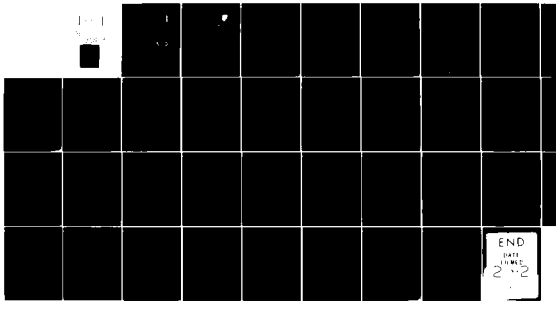
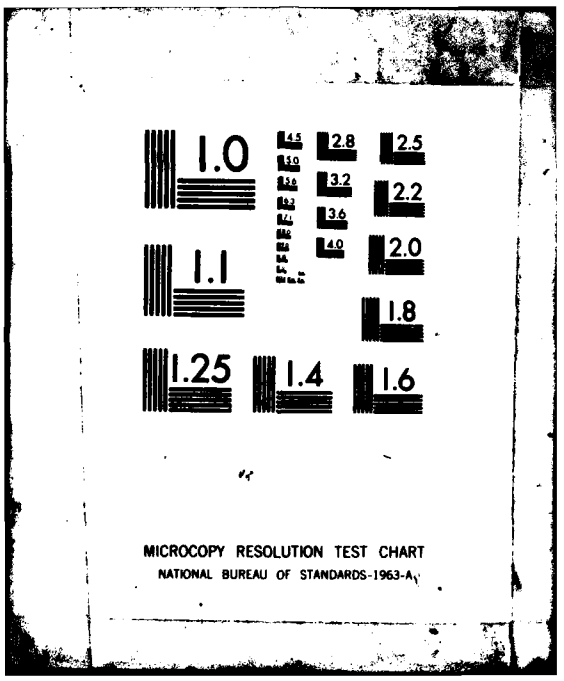
AD-A110 003 INTERMETRICS INC CAMBRIDGE MA F/G 9/2
ADA INTEGRATED ENVIRONMENT I COMPUTER PROGRAM DEVELOPMENT SPECI--ETC(U)
DEC 81 F30602-80-C-0291
UNCLASSIFIED RADC-TR-81-358-VOL-5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

PHOTOGRAPH THIS SHEET

AD A110003

DTIC ACCESSION NUMBER

LEVEL

INVENTORY

Intermetrics, Inc
Cambridge, MA

ADA Integrated Environment I Computer Program Development Specification. Intrim Rept., 15 Sep. 80 - 15 Nov. 81 Dec. 81

DOCUMENT IDENTIFICATION

Contract F30602-80-C-0291 RADC-TR-81-358, Vol IV

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DISTRIBUTION STATEMENT

ACCESSION FOR		
NTIS	GRA&I	☒
DTIC	TAB	☐
UNANNOUNCED		☐
JUSTIFICATION		
BY		
DISTRIBUTION /		
AVAILABILITY CODES		
DIST	AVAIL AND/OR SPECIAL	
A		

DISTRIBUTION STAMP

DTIC COPY INSPECTED 3

DTIC
ELECTE
S JAN 25 1982 D
D

DATE ACCESSIONED

82 01 12 014

DATE RECEIVED IN DTIC

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDA-2

DOCUMENT PROCESSING SHEET

DTIC FORM OCT 79 70A

RADC-TR-81-358, Vol V (of seven)
Interim Report
December 1981

ADA INTEGRATED ENVIRONMENT I COMPUTER PROGRAM DEVELOPMENT SPECIFICATION

Intermetrics, Inc.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This document was produced under Contract F30602-80-C-0291 for the Rome Air Development Center. Mr. Don Roberts is the COTR for the Air Force. Dr. Fred H. Martin is Project Manager for Intermetrics.

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-81-358, Volume V (of seven) has been reviewed and is approved for publication.

APPROVED: Donald F. Roberts

DONALD F. ROBERTS
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-81-358, Vol V (of seven)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ADA INTEGRATED ENVIRONMENT I COMPUTER PROGRAM DEVELOPMENT SPECIFICATION		5. TYPE OF REPORT & PERIOD COVERED Interim Report 15 Sep 80 - 15 Mar 81
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s)		8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0291
9. PERFORMING ORGANIZATION NAME AND ADDRESS Intermetrics, Inc. 733 Concord Avenue Cambridge MA 02138		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62204F/33126F 55811908
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COES) Griffiss AFB NY 13441		12. REPORT DATE December 1981 13. NUMBER OF PAGES 46
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Donald F. Roberts (COES)

Subcontractor is Massachusetts Computer Assoc.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada	MAPSE	AIE
Compiler	Kernel	Integrated environment
Database	Debugger	Editor
KAPSE	APSE	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Ada Integrated Environment (AIE) consists of a set of software tools intended to support design, development and maintenance of embedded computer software. A significant portion of an AIE includes software systems and tools residing and executing on a host computer (or set of computers). This set is known as an Ada Programming Support Environment (APSE). This B-5 Specification describes, in detail, the design for a minimal APSE, called a MAPSE. The MAPSE is the foundation upon which an

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

APSE is built and will provide comprehensive support throughout the design, development and maintenance of Ada software. The MAPSE tools described in this specification include an Ada compiler, linker/loader, debugger, editor, and configuration management tools. The kernel (KAPSE) will provide the interfaces (user, host, tool), database support, and facilities for executing Ada programs (runtime support system).

TABLE OF CONTENTS

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

1.0 SCOPE

1.1 Identification.

This specification establishes the requirements for performance, design, test, and qualification of MAPSE program integration facilities. These include both the set of computer program modules that perform program integration functions and the Ada program library, whose structure and contents facilitate their performance.

1.2 Functional Summary.

The requirements section of this document is divided into four major sections:

1. a design for an Ada program library that supports access by both MAPSE tools and user-written APSE programs;

2. a high-level design for the Ada compiler;

3. a design for a program library linker; and

4. a specification of auxiliary functions required to build, use and maintain the program library.

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

2.0 APPLICABLE DOCUMENTS

Please note that the bracketed numbers preceding the document identification is used for reference purposes with the text.

2.1 Government Documents

[G-1] Reference Manual for the Ada Programming Language, proposed standard document, U.S. Department of Defense, July 1980.

[G-2] Requirements for Ada Programming Support Environment, "STONEMAN", February 1980, Department of Defense.

[G-3] Statement of Work for Ada Integrated Evironment, PR No. B-0-3233, December 1979.

2.2 Non-Government Documents

[N-1] An Incremental Programming Environment, Peter H. Feiler and Raul Medina-Mora, Dept. of Computer Science, Carnegie-Mellon University, April 1980.

[N-2] Diana Reference Manual, G.Goos and Wm. A. Wulf (editors), Institut Fuer Informatik II Universitaet Karlsruhe and Carnegie-Mellon University, 1981.

[I-1] System Specification for Ada Integrated Environment, Type A, Intermetrics, Inc., March 1981, IR-676.

Computer Program Development Specifications for Ada Integrated Environment (Type 5):

[I-2] Ada Compiler Phases, IR-677

[I-3] KAPSE/Database, IR-678

[I-4] MAPSE Command Processor, IR-679

[I-5] MAPSE Generation Support, IR-680

[I-6] MAPSE Debugging Facilities, IR-682

[I-7] MAPSE Text Editor, IR-683

[I-8] Technical Report(INTERIM) IR-684

3.0 REQUIREMENTS

In the MAPSE environment, program integration has two major objectives: support for separate compilation, and linking the separately compiled units. In most environments, even those that support separate compilation, source compilation is considered to be an isolated step, performed prior to program integration; program integration is left entirely to a linker program. However, Ada has been carefully designed to include separate compilation features as an integral part of the language. Implicit in these language features is the assumption that substantial integration occurs at compilation time. The design of the program library, described in detail below, fully supports both the separation and the integration inherent in the separate compilation features of Ada.

The MAPSE program integration facilities are based on the concept that it is the compiler which integrates compilation units into the library. Having the compilation units already integrated by the compiler simplifies the design and implementation of the MAPSE tools, as well as other programs written to access the program library.

Access to the program library is mediated by the virtual memory management (VMM) system, a general purpose system described in [I-5]. The VMM facility allows programs to access efficiently multi-billion byte databases through a window which can be small enough to fit in 16 bit minicomputer memory. Thus the MAPSE can be rehosted to small computer systems with a minimum amount of effort.

The program library makes use of the general database facility of the KAPSE [I-3]; its structure as a composite object containing simple objects is a particular instance of a more general KAPSE facility.

3.1 Program Definition

The major functions of program integration are source compilation and linking. Source compilation is the processing of a text file, containing one or more compilation units, by the Ada compiler. Linking is the processing of a set of compilation units, in a program library, by the Linker to produce an executable representation of those units.

During both source compilation and linking, other processing occurs automatically, as needed, to:

1. create and initialize a program library;

2. recompile units that are inconsistent;

3. create and compile implied but missing units.

Although program development can proceed using only the source compiler and the linker, auxiliary functions relating to the program library are available to the user. These auxiliary functions allow the user to control otherwise automatic functions (such as stub or preamble generation), or provide additional functions not provided automatically.

3.2 Detailed Functional Requirements

Since the design selected for the program library is the basis for all MAPSE program integration activities, that design is presented in some detail below. It should be pointed out that this design is made possible by the VMM, as described under section 3.2.1.3.

3.2.1 Program Library

The universe within which the integration of Ada compilation units occurs is the program library. Within this universe are contained pieces of the "whole program" which is in some stage of development. The program library is designed so that it can maintain a set of compilation units in a well-defined state no matter how complex their interrelationships are or how drastically they may change.

This well-defined state is independent of changes made outside its universe. Multiple Ada program libraries within one MAPSE allow parallel development of unrelated programs without conflict or unwanted interactions.

The correct design and implementation of the Ada program library is one of the keys to achieving the goals of the MAPSE. Of particular importance are choices made in the representation of Ada programs and especially in the implementation of that representation in the light of separate compilation. Those choices influence not only the internal design of the MAPSE/APSE tools, but also shape the user's view of the system.

3.2.1.1 Program Library Requirements

To satisfy MAPSE requirements, a program library must support:

1. efficient forms of representation of an Ada program during the process of compilation;

2. permanent retention of and access to those forms that are needed by other tools; that is, the compiler (when referencing separately compiled units or when recompiling a unit automatically); the linker; the debugger; and yet-to-be-conceived APSE tools;

3. addition (to those permanent forms) of as yet unspecified information produced by unspecified APSE tools.

Most importantly, the fact of separate compilation should introduce as few complications and as little overhead as possible to the fulfillment of these requirements. To achieve the required simplicity and provide appropriate support for separate compilation, the program library must be viewed globally rather than locally, as discussed below.

A local view of the program library might, for example, be one in which the compiler produces a representation of each compilation unit that is completely self-contained. This would require the copying of information from the representation of other compilation units. An obvious drawback here is the space overhead for many copies of the representation of an imported type or specification, one in each referencing unit. With certain styles of programming-in-the-large, the space and time overhead of copying can be severe.

More serious is the impact this view has on the design of the MAPSE tools and the integrity of the program library. A tool that must look at the representation of more than one unit has to integrate the separate self-contained worlds. It must keep what has been defined by each unit, throw out what has been copied, and insure that the multiple copies are self-consistent. A simple example is the context specification WITH B,C where both B and C contain WITH A. Since B and C both contain copies of A the two versions of A must be sorted out. They might be different, in which case a choice must be made. This integration must be performed each time a representation of a set of units is accessed. The integration exists only during program execution.

A global view, on the other hand, is one in which all of the units are integrated together. The compiler produces a representation of a unit that is already integrated with the other units. Instead of copying information from a previous compilation, the information is directly referenced (pointed at). In a completely global view, all traces of the original separation disappear after a unit is compiled, making it impossible to maintain multiple revisions or versions of a compilation unit. To fulfill MAPSE requirements, the global view must be adjusted to provide this capability.

The MAPSE program library described below is a logically integrated yet physically distributed world. Each compilation unit is represented separately in it own space (file), yet is integrated

with the other units. This approach avoids the problems of both the local and the non-distributed global views. It avoids the overhead, complexity, and cost associated with copying. At the same time, it allows multiple versions of units to coexist in the same program library. The separateness of compilation units remains highly visible.

Two things are fundamental to the successful implementation of this view. One is the method of representing the separation; the other is the method of supporting references across that separation. Each method is discussed in the following two sections.

3.2.1.2 The View from the KAPSE

This section describes the program library in terms of its constituent parts of the KAPSE database. The description emphasizes the separateness of compilation units; the integration aspects are discussed in the next section. It is important to point out that the KAPSE facilities used to support the program library are general purpose; they are not "wired into" the KAPSE design.

This section uses some KAPSE database terminology that may be unfamiliar to the reader. For more information about these terms, see the KAPSE Database specification document [I-3]. To aid the reader, however, the term "distinguishing attribute" refers to a label whose value helps determine the name of an object. The name of an object is the set of all of its distinguishing attribute-value pairs. Objects may also have non-distinguishing attributes; these are labelled values as well. A set of objects may be partitioned (a subset selected) by means of the values of either distinguishing or non-distinguishing attributes. In the sections that follow, distinguishing attributes are indicated with an asterisk (*).

The program library is a composite object containing objects distinguished by the attribute PER. The values for the PER attribute are:

*PER=>(SOURCE,COMPILATION,UNIT,LINK,LIBRARY)

Thus the program library has five component objects. The category of each of these components is described in the following sections.

(a) PER=>SOURCE. This component has a category which may be defined by the user to contain the source text for the program library. The design of the program library and the compiler does not dictate the organization or location of Ada source text. The PER=>SOURCE component may be absent, may be a simple object, or may be a

composite object structured as the user sees fit. When the user submits source text to the compiler, the specified name is first interpreted relative to the PER=>SOURCE component of the specified program library. If a text file is not so located, the specified name is interpreted relative to the current context (as are all names).

(b) PER=>COMPILATION. This component is a composite object containing simple objects. One such simple object is created for a given submission of text to the compiler, no matter how many compilation units are contained therein. The set of attributes for these objects are:

```
*COMPILATION=>submittal number
 OPTIONS=>options string
```

The creator of these objects is the first phase of the compiler, LEXSYN, which performs lexical and syntactic analysis of the source text. The object contains the abstract syntax tree with lexical attributes, a record of lexical and syntactic errors, and a compressed representation of the source text for the purpose of generating listings. All of the compilation units submitted together are represented together. Once created, this combined representation is used as a read-only input to the remaining phases of the compiler, which process only one unit at a time. Since it is not modified, it can be and is reused for unit recompilation. This saves the overhead of lexical scanning and parsing (especially if there were multiple compilation units), and guarantees that the unit is recompiled exactly as it was given in its source compilation. Thus changes to the unit's original source text or to its PRAGMA INCLUDE files do not affect the accuracy of unit recompilation. The object contains no semantic information; its contents are derived solely from the source text without reference to the rest of the program library.

This design supports the extension of the MAPSE tool set to include a syntax-directed editor [N-1] that creates and edits a program directly in its parse tree form, unparsing the tree during user interactions to produce a readable form. Such an APSE tool could easily be built with no change to the program library design, and only a minor change to the MAPSE compiler. It would create the PER=>COMPILATION object directly, bypassing the LEXSYN phase of the compiler.

The OPTIONS attribute records the options specified by the user when the source was compiled; this is used during unit recompilation to supply the same options.

(c) PER=>UNIT. This component is a composite object containing simple objects. Each simple object is created to represent information relating to a single compilation unit, regardless of how many are submitted

together in one source compilation. They are created for both source compilation and unit recompilation. The distinguishing attribute CONTENTS=>(DIANA,LIST) defines the permanent outputs of the remaining phases of the compiler, semantics through listing generation.

The complete set of attributes for PER=>UNIT are

```
 UNIT=>identifier
 SUBUNIT=>identifier or null
 PART=>(SPECIFICATION,BODY)
*CONTENTS=>(DIANA,LIST)
*VMSD=>number
 KIND=>(PACKAGE,PROCEDURE,FUNCTION,TASK)
 COMPILATION=>number
 RECOMPILATION=>number
 LINKREFS=>number
 VERSION=>string or null
 SHARED_LIB=>string or null
 USAGE=>(NEW,USED,OUT_OF_DATE,MAP)
 ERRORS=>(NONE,MILD,SEVERE)
```

The object with CONTENTS=>DIANA contains all of the permanent forms of representation of semantic analysis and code generation for a compilation unit, including information of interest to the debugger. It is an integrated form of all of the traditional outputs of a compiler: object module, program tree, symbol table, debugger table. It is created during semantic analysis by extracting the syntax tree for a single compilation unit from the PER=>COMPILATION object. Semantics and subsequent phases of the compiler add additional information to the extracted tree. Once created, it is not modified as a result of its later use by either the compiler, linker, or the debugger.

The object with CONTENTS=>LIST contains a printable listing for the compilation unit. One LIST object is created for each compilation unit, allowing unit recompilation to also produce a listing. Although the source representation of a unit does not change as a result of unit recompilation, the semantic analysis of the unit may change. As a result of unit recompilation, semantic errors may appear or disappear, cross reference information may differ, etc.

The CONTENTS=>LIST object is produced by the last phase of the compiler, the LISTER. All of the information that the lister needs to generate a listing has already been permanently saved, so the compiler may be run with the LIST option turned off. At any later time the LISTER phase may be run separately to produce a listing. Thus a project manager may choose to either save compilation unit listings at compile time, or produce them only as needed.

The value of the UNIT attribute is the identifier of the library unit as it appears in the source text, but normalized by capitalization. If the compilation unit is a library unit, the value of the SUBUNIT attribute is the null string. If the compilation unit is a subunit, the value of the SUBUNIT attribute is the identifer of the separate body (again normalized), while the

value of the UNIT attribute is the same as its ultimately enclosing library unit. (Intermediate subunit names are not required as attributes, since the language requires all of the subunits, sub-subunits, etc. of a given library unit to have distinct identifiers. Thus SEPARATE A.X.B and SEPARATE A.Y.B is illegal in Ada because the B sub-subunit identifier is not unique.)

The value of the PART attribute serves to differentiate between the separate compilation of the specification and body of a package or subprogram. In the case of a package the PART=>BODY object may be absent without error. In the case of a subprogram, the PART=> SPECIFICATION object may be absent without error, in which case the PART=>BODY object defines the specification as well.

The value of the VMSD attribute is a special number which is the basis for the integration of compilation units in a program library. It also serves to distinguish between different versions and revisions of the same compilation unit. The actual value of the number is not directly significant to the user. Its significance to the design of the program library and to the method of access is discussed in the next section. (The attribute label VMSD is an acronym for Virual Memory Sub Domain.)

The remaining attributes are explained in section 3.2.2.2 on compiler processing.

(d) PER=>LINK. The PER=>LINK component is a composite object containing program context objects. The program context objects are created by the linker. The complete set of attribute are:

```
*CALL=>identifier
 MAIN=>identifier
```

The program context object contains the initial memory images (pure and impure) as built by the linker. It also contains a machine readable map summarizing the results of the link: identities of each compilation unit included in the link and the relative locations of each control section (defined later) in the executable image. The program context also contains, by convention, a window to the program library that created it (and in which it is originally built). This allows the program context to be copied outside of the program library without losing the connection with its originating program library.

The value of the MAIN attribute is the identifier of the "main" compilation unit. The value of the CALL attribute is the name by which the executable program will be invoked, and defaults to the MAIN name.

(e) PER=>LIBRARY. The PER=>LIBRARY component of the program library is a composite object. Objects in the PER=>LIBRARY composite object relate to the library as a whole. The category of this composite object is not pre-defined. It may contain objects structured as the user sees fit. However, its attributes serve to record the status of the library, as follows:

COMPILATIONS=>last submittal number
LAST_VMSD=>last assigned VMSD number
REVISIONS=>number of revisions to keep

The value of the COMPILATIONS attribute is incremented for each source compilation. It need never be reset, and is used to generate the value of the COMPILATION attribute for PER=>COMPILATION objects. The value of the LAST_VMSD attribute is used to generate the VMSD attribute value for objects with PER=>UNIT, explained in section 3.2.2.2.

The REVISONS attribute helps control the deletion of OUT_OF_DATE revisions.

3.2.1.3 The View from the MAPSE

There are many questions regarding the representation of Ada programs.

What aspects of the program does it represent? How is the representation defined? How are instances of the representation created, saved, and later accessed? The previous section discussed the external database aspects of the representation. What about the contents of those "simple objects"? What does it look like at the "bit" level? How does a program "see" it? It is to these questions that this section is addressed.

There is nothing unusual in the idea of an Ada program that can access representations of Ada programs, including itself. Any program that can read a text file can read a simple representation of itself. The representations in an Ada program library are more than text representations, however. They have been processed by a compiler, which transforms the source text of a program into a more easily manipulated form (lexical and syntactic analysis), analyzes the program's meaning (semantics), and translates it into a form which is understood by the target computer hardware (code generation). All of these processes produce information that is derived from the original source text representation using additional knowledge (either built into or accessible to the compiler) which represent the rules of Ada and the target machine.

(a) The Diana Abstraction. Diana is a form of intermediate representation for Ada programs that has been suggested for adoption as a standard. It is intermediate in that it represents only the information produced by a compiler after the completion of semantic analysis. (The preliminary Diana document [N-2] suggests an approach to representing code generation information. This document is attached as an appendix to the compiler phases B5 specification [I-2]). In addition to being only an intermediate representation, Diana is also an abstract representation. Rather than explicitly describing the

data structures of the representation (as record types, for example), Diana gives only an abstract type, called TREE. Diana exists (in one form) as as Ada package specification that defines the type TREE as PRIVATE, and then defines many procedures and functions to operate on that type. This is a very useful abstraction, and allows the compiler to create (and other tools to access) the Diana representation of an Ada program without depending on the particular implementation of that representation. Our design is based on this technique.

However, PRIVATE types must ultimately be defined! Choices must be made in the implementation of the DIANA representation. Although the abstraction isolates the parts of the compiler that manipulate the tree, the design of the compiler as a complete tool, its portability , and especially the demands it makes on computing resources, all depend quite heavily on the implementation, not the abstraction. Furthermore, the abstraction treats lightly or ignores entirely some very serious issues for a practical MAPSE compiler, such as address space limitations, efficient permanent retention of the representation, and separate compilation. The DIANA implementation must face these issues.

(b) The Diana Implementation. Our implementation of Diana rests upon the virtual memory management (VMM) system, described in detail in a separate document [I-5]. This system is a general-purpose facility that may be used by any Ada program in an APSE, for purposes not limited to representations of Ada programs. The system has a definitional facility (using Ada) to define arbitrarily complex typed data structures. The data structure is defined in terms of virtual records (analogous to Ada records) that have components of a user specified type. One type of component is used as a reference type (analogous to an Ada access type), through which the virtual records may be joined into trees, lists, and other complex structures. Note: The word "node" will be used interchangeably with virtual record. This reflects the fact that the Diana nodes are implemented in terms of virtual records. However, a Diana node is a logical construct which may be represented by more than one virtual record.

The other part of this system is an access facility that:

1. allows a program to create and manipulate instances of the data structures previously defined;

2. has a built-in capability to permanently retain the data structures for later use or modification, as desired, in any of three forms: an efficient binary form (the same one as created), a human-readable ASCII form, and a compressed binary form that is machine independent like the ASCII form, but very compact;

3. allows the creation of and access to extremely large data structures which exceed the memory space of the computer;

4. supports use of the reference (VMM locator) type to data structures created at separate times and in separate spaces (files) as easily and efficiently as within the same space.

The permanent external binary representation is typically what is created and accessed by a program. With this form there is no need for a "read" phase prior to access of the representation. Since no transformation of a node occurs on access, the usual "read" phase, in which every node of a data structure is processed prior to any access, is totally unnecessary. The savings are enormous for a program that needs access to only a part of the data structure.

The support of cross-file references is fundamental to the integration of separate compilation units. The virtual memory management system defines the concept of a domain, which is a virtual memory address space. A domain contains one or more subdomains (virtual memory files). Each compilation unit is assigned a subdomain number (the value of its VMSD attribute). Virtual memory locators contain a subdomain number and an offset (in multi-byte units) within the subdomain (file). Virtual memory locators can refer to locations within the same subdomain or to locations in other subdomains of the same domain. The locator is context (subdomain) independent, so it may be copied from one subdomain (compilation unit) to another without loss of meaning. If references in a program library did not have this property, complex translation rules would have to be applied at difficult-to-determine times. This would complicate not only the compiler, linker, and debugger, but every APSE tool that must access the program library.

The value of the VMSD attribute is not of direct significance to the user. It defines the unit's integration slot in the program library. It also allows multiple revisions or versions of a compilation unit to coexist, because they have different VMSD values. No two PER=>UNIT objects in a program library have the same VMSD number (except the matched pair of CONTENTS=>DIANA and CONTENTS=>LIST). The number is limited to a maximum per domain (suggested limit=4K, but a larger limit can be designed) and hence limits the number of distinct units and revisions of units in a single program library. When the LAST VMSD number of a program library reaches its limit, it is reset so that new VMSD numbers can be assigned from the "holes" left by deleted objects.

It is desirable (but not necessary) to contain a virtual memory locator in 32 bits. Allocating 12 bits to a subdomain number and 20 bits to a subdomain offset (in 16 byte units) yields the following limits for a program library:

4096 compilation units (including multiple versions and revisions)

16,777,216 bytes per compilation unit

A different allocation of a 32 bit virtual memory locator yields a greater number of smaller units, or a lesser number of

bigger units. Allocating 64 bits increases the limits for both the number and size of compilation units to unimaginably large values.

The PER=>COMPILATION object produced by the LEXSYN phase of the compiler requires representation of only lexical and syntactic attributes. Therefore, the nodes used in the abstract tree may be smaller versions than those in the PER=>UNIT,CONTENTS=>DIANA object. The normalizations or transformations of the Diana tree required by semantic analysis have not been performed in the abstract tree.

(c) Global Cross References. The representation of global cross references presents an interesting problem. The use of a global symbol (defined in a different compilation unit) represents no problem, because cross-unit references are directly supported by the VMM system. The identity of the global symbol is the VMM locator to its definition. However, it is desirable to record all references to a symbol as a list of cross reference entries attached to the symbol definition. This cannot be done for global references without modifying the file containing the definition; such modification has been ruled out, because it is desirable to keep the results of a compilation unmodified by its use. (The indication of its use is recorded in the USAGE attribute, but this is a modification of the attribute, not its contents.)

The solution is to create a global reference VMM locator association set in each unit, with one set element for each global symbol referenced in the unit. The locator associations in this set have key values that are the VMM locators for global symbol definitions and associated values that are locators for the lists of references. A program to gather, process, or search through all references to global symbols can be easily designed and will perform efficiently.

3.2.1.4 Summary of Program Library Attributes

Note: distinguishing attributes are marked with an asterisk (*).

```
*PER=>LIBRARY                  --for each program library
   COMPILATIONS=>number        --number of compilations into this
                                 library
   LAST_VMSD=>number           --last assigned value of VMSD
                                 attribute
   REVISIONS=>number           --number of revisions of a unit
                                 to keep

*PER=>COMPILATION            --for each source submittal
   *COMPILATION=>number        --value of PER=>LIBRARY COMPILATIONS=>
```

```
   OPTIONS=>string              --options used at submittal

*PER=>UNIT                      --for each compilation unit
   UNIT=>identifier             --Ada name
   SUBUNIT=>identifier|null     --subunit Ada name or null
   PART=>(SPECIFICATION,BODY)
                                --differentiates the 2 parts of units
   *CONTENTS=>(DIANA,LIST)      --distinguishes the 2 permanent
                                  outputs
   *VMSD=>number                --virtual memory sub domain
   KIND=>(PACKAGE,PROCEDURE,FUNCTION,TASK)
                                --kind of compilation unit
   COMPILATION=>number          --identifies the source submittal
   RECOMPILATION=>number        --number of times recompiled
   LINKREFS=>number             --records usage as linked unit
   VERSION=>string or null      --optional name of version
   SHARED_LIB=>string or null
                        --if not null, name of library containing
                        --the actual contents; object in this
                        --library actually is empty
   USAGE=>(NEW,USED,OUT_OF_DATE,MAP)
                        --status of usage
   ERRORS=>(NONE,MILD,SEVERE)
                        --record of compilation errors

*PER=>LINK                      --for each link
   *CALL=>string                --name used to call program
   MAIN=>identifier             --name of main program unit
```

3.2.2 Ada Compiler

The compiler is described in this document down to the phase level only. The detailed description of each of its phases appears in a separate document [I-2].

3.2.2.1 Inputs

The user's request to compile Ada source is represented as:

```
COMPILE [SOURCE=>text_file][LIB=>prog_lib][options]-->summary
```

The text file containing the source is identified by the optional SOURCE=> parameter. If the SOURCE=> parameter is not specified, the source is read from the standard input file.

The specification of the source text is intentionally unrestricted. As a consequence, management of Ada source text is not inextricably built into an Ada program library. The level of source text control is the choice of the user or project manager. The source history system may be used, as described in the KAPSE B5 specification document [I-3]. This may be used for non-Ada text (e.g. documents) as well as for Ada source; inventing an Ada source management system as an inseparable part of the program

library is unnecessary. By reading from STANDARD_INPUT, the source compiler simplifies its connection to other tools, allowing pipelines into the compiler, syntax checking from the editor, etc.

The program library that is to contain the results of the compilation is specified as a string (here "prog_lib"). The string may name a pre-existing program library, or it may name a new one to be created. If the program library is not specified at all, the COMPILE request is interpreted as a request for a syntax check with no semantic processing and no permanent output. (An MCL script which provides a compile, link, and go capability can identify a program library as a temporary object.) In any case, the summary is written to the default output file (STANDARD_OUTPUT). The summary reports what units were compiled and lists any errors may have occurred. (Errors appear in the listings as well.)

Options may be specified with the COMPILE request. If the user desires to retain multiple versions of a compilation unit, the VERSION option is used to assign an arbitrary string to identify the compilation unit(s) produced. The USE_VERSIONS option may be used to select one of several possible versions of a referenced unit. Referenced units are:

1) the specification corresponding to the body being compiled;

2) the enclosing unit(s) of a subunit;

3) the specifications of units named in WITH statements;

4) the bodies of referenced specifications which are INLINE or GENERIC.

The USE_VERSIONS may specify a list of version identifications. The version (unit) selection algorithm is described in section (3.2.2.2).

The DEBUG option affects the generation of code to support the debugger's control of program execution at the statement level. The possible values are DEBUG=>ON (the default) and DEBUG=>OFF. If the debugger encounters a compilation unit compiled with DEBUG=>OFF, breakpoints may not be set at arbitrary statements, only at higher level constructs such as subprogram invocation, tasking interactions, exceptions, etc.

The OPTIMIZE option affects the level of optimization with respect to debugging capabilities. (The OPTIMIZE PRAGMA tells the compiler what to optimize -time or space.) Since sophisticated optimization can affect the validity of debug time interactions, the user can control the effect optimization has on debugging as follows:

```
OPTIMIZE=>CAN_MODIFY  -- The compiler must not move a fetch of
                      -- variable past a statement boundary;
                      -- the user can modify the value of a variable
                      -- at a breakpoint and expect the modified
                      -- value to be used in subsequent execution.

OPTIMIZE=>CAN_INSPECT -- The compiler must not move a store
                      -- into a variable past a statement boundary;
                      -- the user can display the value of a
                      -- variable at a breakpoint and expect to see
                      -- the value as used in subsequent execution.

OPTIMIZE=>ON          -- All optimizations enabled; the user
                      -- may inspect or modify variables, but the
                      -- results may be misleading; meaningful
                      -- debugging is possible by consulting the
                      -- generated code listing.

OPTIMIZE=>OFF         -- No optimization. (No additional debug
                      -- capabilities compared with CAN_MODIFY.)
```

The options controlling the generation of a listing are:

```
LIST=>OFF             -- No listing is produced; a listing may
                      -- be requested at a later time.

LIST=>ON              -- A full listing is produced with all
                      -- sections present (the default).

LIST=>(SOURCE,INCLUDE,XREF,ATTRS,ASM,STATS,ENV)

                      -- One or more of these options may be specified
                      -- Each controls the section of listing as
                      -- described in the LISTER section of the
                         compiler phases document [I-2].
```

The following option applies to a compilation containing more than one compilation unit.

```
REORDER=>(YES,NO)
```

If REORDER=>YES is specified (the default), the compilation units may be presented in any order. After processing by LEXSYN, the compilation units are reordered, if necessary, to obey the Ada language rules on order of compilation (G-1, 10.3). The reordering

eliminates the burden, otherwise placed on the user, of knowing the correct order of compilation units. If REORDER=>NO, the units are compiled in the order given.

3.2.2.2 Processing

The compiler performs most of its processing in twelve phases. These phases are described in detail in a separate document [I-2]. This section describes the processing done outside these phases: the preparation of the program library, the sequencing of the phases, and the preparation of input to and output from each phase. For the purpose of this section, the phases of the compiler fall into two categories:

1) LEXSYN, which performs lexical and syntactic analysis of a compilation (which may contain more than one compilation unit);

2) the remaining phases, which process a single compilation unit (SEM,GENINST,STATINFO,STORAGE,EXPAND,FLOW,VCODE,TNBIND, CODEGEN, FINAL, LISTER).

For a given invocation of the compiler, the LEXSYN phase is invoked only once. The remaining phases (as a group) are invoked once for each compilation unit.

The processing by the compiler may be summarized as follows:

1. Options processing

2. Program library processing

3. Source processing

4. Unit processing

Options processing starts by using a KAPSE call to get the parameters specified by the user. This is returned as a single string that is saved as the OPTIONS attribute of the PER=>COMPILATION object (see below). The LIB and SOURCE values are extracted from the string at this time.

Program library processing starts by creating a program library if the specified library does not yet exist. This is done by invoking the program library generation tool, described in section 3.2.4. The COMPILATIONS attribute of the PER=>LIBRARY object is modified by increasing its value by one. A VMM subdomain file is created in the PER=>COMPILATION object to hold the results of source processing (the abstract syntax tree). It is given the appropriate attribute values. The attribute COMPILATION is given a value corresponding to the current value of the COMPILATIONS attribute of the PER=>LIBRARY object. The OPTIONS attribute is given the value as saved from options processing. If a program library is not specified (syntax checking only), this object is merely a temporary

object in the invoker's program context. Note: the subdomain number is always zero, since the PER=>COMPILATION objects do not participate in the PER=>UNIT integration.

Source processing is performed by the LEXSYN phase. The default input file is assigned, if necessary, using the SET_INPUT procedure of TEXT_IO. If a name is specified via the SOURCE=>name parameter, an attempt is made to OPEN a file of the name "prog_lib.SOURCE.name". If this fails, the file "name" is opened. The LEXSYN phase is invoked, passing it the PER=>COMPILATION (or temporary) object to contain its output. (In the case of syntax checking, the compiler's final action is to pass the temporary object produced by LEXSYN to the LISTER. All semantic and code generation phases are bypassed. The LISTER's output in this case consists only of lexical and syntactic errors.)

Unit processing starts by inspecting the output of LEXSYN, in the PER=>COMPILATION object. The compilation units are first reordered if REORDER=>YES. The processing as described below is repeated in a loop for each compilation unit which LEXSYN created. The parts of unit processing are:

1. referenced unit processing, which selects a particular version/revision of a unit;

2. phase processing, which invokes the remaining phases of the compiler;

3. attribute processing, which modifies attributes of PER=> UNIT objects.

The highest part of the abstract syntax tree for the compilation unit is inspected to determine both the context specification (WITH statements) and the kind of unit (library unit specification, library unit body, or subunit). This analysis produces the referenced unit list consisting of Ada unit and subunit identifiers. The unit selection algorithm is applied to each member of this list to determine the values of the VMSD attribute of the (PER=>UNIT,CONTENTS=>DIANA) objects to be used as the referenced units. As explained in the special requirements section below, this algorithm insures selection of consistent units by performing unit recompilation if necessary. Thus referenced units may be recompiled prior to their use by the original unit being compiled. This is a recursive process, since the referenced units themselves may refer to units that require recompilation, and so on. The recursion does not occur inside any of the compiler phases, however, but rather in the unit processing section of the compiler. Once the remaining phases of the compiler are invoked for a given unit, they have a consistent set of referenced units already available, so that recursion within the phases is unnecessary.

After the selection algorithm has produced a referenced unit list in terms of the VMSD attribute of CONTENTS=>DIANA objects, a new CONTENTS=>DIANA object is created to contain the results of the given unit compilation. The VMSD attribute of the new object is

assigned by incrementing the value of the LAST_VMSD attribute of the PER=>LIBRARY composite object. If the number obtained is in use because a PER=>UNIT object with that VMSD attribute value already exists, the LAST_VMSD number is incremented until an available number is found. On reaching its limit, the number is reset to 1. This new object is initialized with the list of (read-only) referenced objects, the identity of the PER=>COMPILATION object (also read-only), and the identity of the compilation unit within that object which is to be processed. (The list of referenced objects may be used by any program which later wishes to access the DIANA file of a given compilation unit.) The new CONTENTS=>DIANA object is then passed to the remaining phases of the compiler.

All of the remaining phases (except the LISTER) are always invoked, whether or not they must perform any actions. The phase itself decides what processing must be performed. The LISTER is called conditionally on the listing option. It writes to the default output file, which is directed (via TEXT_IO.SET_OUTPUT) to the appropriate PER=UNIT, CONTENTS=>LIST object. The LISTER may be called independently of the COMPILER, in which case its output may be directed anywhere.

After phase processing for a unit is complete, the attribute processing part of unit processing occurs. This sets the USAGE attribute of each referenced unit to USAGE=>USED. This indicates that references into those units may exist. The USAGE attribute of all other revisions of the same version of the just compiled unit is set to OUT_OF_DATE. This indicates that any references to the previous definition should be updated. If the USAGE attribute of a revision (which is about to be set to OUT_OF_DATE) is UNUSED, the revision is deleted instead. The OUT_OF_DATE value of the USAGE attribute may be used to control deletion of units by the user. Automatic deletion of OUT_OF_DATE units may be requested by the user via the REVISIONS attribute of the PER=>LIBRARY object; this causes deletion of the oldest revision when the number of revisions exceeds the specified limit. Normally, automatic deletion is not performed to allow a comparison to be made between revisions (see special requirements below), and to allow analysis of the units which still refer to the OUT_OF_DATE revision.

3.2.2.3 Outputs

The outputs of the compiler are the compilation summary written to the default output file, and the program library. The effect on the program library in a logical sense consists of defining or redefining the compilation units supplied in the source text and redefining the automatically recompiled units. The effect in a physical sense is the creation of new files. These files are described in section 3.2.1.2. The operations performed on these files by each phase is described in the compiler phases B5 specification document [I-2].

3.2.2.4 Special Requirements:

The source processed by the compiler may redefine a compilation unit which had been previously compiled and then referenced by another compilation unit (via the WITH statement, for example). The redefinition implies that all referencing units must be recompiled to be affected by the change. Until the affected units are recompiled, there exists an inconsistency in the program library. However, it is an unwelcome burden on the user to require the resubmission of the source of an unchanged compilation unit merely because a unit which it references has been altered. The method by which the compiler and linker lift this burden from the user is detailed in the following paragraphs. (For an example, see Section 6.)

(a) Unit Recompilation. A program library can be well-defined and yet be inconsistent, as long as the inconsistency is properly represented. For example, a specification might be changed without recompiling units that depended on the original. The units requiring recompilation are inconsistent until they are recompiled using the changed specification. The term "unit recompilation" refers to recompilation of a defined unit (usually to make it consistent) by effectively using the same source text as was submitted in a source compilation, but without requiring the user to actually resubmit a source text file. This is possible because the PER=>COMPILATION object, which has saved the output of the first phase of the compiler, may be used to recompile any of the units therein. Recompilation therefore bypasses the lexical and syntactic analysis performed by LEXSYN. A single unit will undergo unit recompilation separately even though it was submitted together with other units in a source compilation. Unit recompilation may be requested explicitly by the user with the RECOMPILE program, although it is rarely necessary to do so. Unit recompilation will occur automatically according to the following two principles:

1) (Ease of use): The user never has to remember what or when to recompile. The system will always recompile whatever and whenever necessary.

2) (Efficiency): Automatic unit recompilation is delayed until use of an inconsistent unit would otherwise result.

Both the source compiler and the linker will perform automatic unit recompilation according to the above principles. These principles avoid wasted recompilations and allow the user to interact in various ways with the program library prior to the "domino effect"*; that is, the widespread recompilations made necessary by a change to a low level specification. The change must be propagated by recompiling all affected units (those which

*A single line of dominoes is an unlikely analogy. To be more accurate, the domino analogy is likely to include both branching and merging. Still, the image of every domino (compilation unit) getting knocked over (recompiled) by a push on (change of) a single one is somehow psychologically apt.

reference the specification), and then those which reference the recompiled units, and so on. By the second principle the unit recompilations do not occur when the low level specification is changed. Instead, they occur when the referencing unit is used (as a referenced unit itself or by the linker).

An obvious example of the savings in recompilations is when the user is about to submit a different source version of an affected unit; prior unit recompilation would be a waste of time. A less obvious interaction with an even greater savings is the use of the MAP program. This program allows the user to assert that two versions/revisions of a compiled unit are identical. The program checks the assertion and if true, maps the old to the new. The mapping allows units compiled using the old version to be consistent with the new without recompilation, thus stopping the domino effect. The VMM system supports this mapping by translating automatically all of the old VMM locators to new VMM locators.

(b) Unit Selection. Unit recompilation occurs in the compiler during the unit processing described above. A key part of this processing is the unit selection algorithm. Given a unit (subunit) identifier and an optional USE_VERSIONS list of version strings, this algorithm selects a particular compiled unit, identified by the value of its VMSD attribute. In the simple case, where the user has not specified the VERSION option for any compilations and therefore has not requested use of a particular version with the USE_VERSIONS option, there is only one version for each unit, with the VERSION attribute a null string. Multiple revisions of this (null) version still will occur by source compilation or unit recompilation. The unit selection algorithm picks the most recent revision which has been successfully compiled (ERRORS=>NONE). Most recent is defined by the highest value of the COMPILATION attribute. (This avoids problems due to operator error in setting the date.) If unit recompilation has occurred, multiple revisons may exist with the same COMPILATION value; in this case the one with the highest RECOMPILATION number is chosen. (As a result of the attribute processing described in a previous section, only the most recent revision has a USAGE attribute whose value is not OUT_OF_DATE. However, the VERSION attribute of a unit may be changed arbitrarily by the user, so this cannot be relied upon.)

Before the candidate unit is made the final choice of the unit selection algorithm, it is tested to see if it requires recompilation. Recompilation of unit A is required if the results of the unit selection algorithm (recursion here) on A's referenced units yields at least one unit that is a different revision than that used when A was compiled. If so, it is recompiled, and the results become the final choice of the (outer) unit selection processing.

By specifying a VERSION option on the COMPILE request, the user indicates that the unit has an additional identity besides its Ada name. The VERSION identity of a unit affects the unit selection

algorithm by means of the USE_VERSIONS option of the COMPILE or LINK requests. This identity may be used at link time to select between alternate bodies for a given specification, or may be used at compile time to select one of several versions to be used as a referenced unit. Recompiling a version produces a revision of that version only, not of other versions. Thus other versions are not marked OUT_OF_DATE or deleted. (The version identity of a unit is not altered by recompilation; however version identities selected for its referenced versions may be different on recompilation.)

The value of the VERSION identification string is arbitrary; it may be changed by the user at any time. Its significance is up to the user, but it affects the actions of the compiler and linker. It may be used to identify a partition in the program library to select units for deletion. A trial compilation may be entered with VERSION=>TEST, for example. The TEST version may be linked in or used as a referenced unit, then later either deleted or reassigned a different version. The simplest use of VERSION is to distinguish between the multiple bodies of a single specification. If the body has no subunits (and is not INLINE or GENERIC), no dependencies on the VERSION occur during compilation; only at link time must a choice be made. This simple use of VERSION will not cause automatic unit recompilation at the time of a COMPILE request. However, if the USE_VERSIONS option causes a choice to be made at compile time, then a change to the option may cause automatic recompilation.

The list of version identities in the USE_VERSIONS list has the following effect on the unit selection algorithm. The existing versions of a unit are checked for a match in the list; the list is ordered in decreasing priority, so the earliest match is selected. If no match is found, the null version is chosen if there is one; if no null version exists, the most recent version is chosen. Thus the null version acts like a "default" version.

3.2.3 Linker

The term "linking" refers to the processing performed by the linker on a program library to produce an executable program context. This is represented by:

```
LINK LIB=>prog_lib [MAIN=>unit_name] [options] --> link summary
```

The naming of a program library is required; the name of a "main program" unit is optional. A short summary of the link is produced on STANDARD_OUTPUT. During the processing of a link request, the linker may perform the following functions automatically:

unit recompilation, if necessary;

stub generation to supply missing bodies for library units or subunits;

preamble generation, if the "main program" has Ada formal parameters.

3.2.3.1 Inputs

(a) Program Library. The program library is specified to the linker with the LIB=> parameter. This library contains one or more compilation units which are to be included in the executable result.

(b) Main unit. The compilation unit which is to be the "main program" is specified with the MAIN=> parameter. This parameter is optional; if it is not specified, a main unit is computed (as described in the processing section below) if there is only one reasonable candidate.

(c) Options. The following options may be specified:

USE_VERSIONS=>list of version identities

The list is used to select between multiple versions of units, as described in the processing section below.

CALL=>name

This is the name to be used to invoke the program context. If not supplied, it defaults to the main unit name.

(d) Linkable Diana . The compilation units in the program library (PER=>UNIT,CONTENTS=>DIANA) contain the Diana representation, with additional information supplied by the code generator. With traditional compilers the compiled machine code and data are written to an object module or "rel" file, separately from any intermediate representation (such as a symbol table). In linkable Diana, the machine instructions and data are integrated with the intermediate representation (the Diana tree). There are a number of advantages to this design. Since the CONTENTS=>DIANA objects are themselves integrated by the compiler, so are the machine instructions and data, thus eliminating some of the work traditionally done by a linker. Also, since the symbolic information and the machine code are in the same file, they are always consistent.

This section describes the representation of the information that traditionally appears in an object module. Each construct is implemented in terms of VMM virtual records, called nodes, as described in the following sections.

The construct of primary concern both to the code generator and the linker is the control section (CSECT). A CSECT is a block of storage units that are to be allocated contiguous space in memory by the linker. The code generator defines the size and function of each CSECT, and also defines any or all of the storage units within. The storage unit is as defined in package STANDARD.SYSTEM for the target machine. For example, on both the 370 and 8/32 target machines, STORAGE_UNIT:=8, which defines the storage unit as 8 bits. The term "SU" will be used in the following sections to mean:

```
Type SU is 0..2**STORAGE_UNIT-1;   --defines a byte for 370, 8/32

For SU'SIZE use STORAGE_UNIT; --length specification
```

The node defining a CSECT has the following components:

1. a list of ENTRY nodes;
2. a list of compiler SU definition (CSUD) nodes;
3. a list of linker SU definition (LSUD) nodes;
4. the CSECT's function (spec elaboration, body elaboration, body call, etc.);
5. the size of the CSECT (in SUs);
6. an alignment (in SUs)-the linker will place the CSECT at an address divisible by this number;
7. a pure vs. impure indicator (pure CSECTs are subject to memory protection and sharing at load time);
8. a VMM locator for the defining Diana construct.

An ENTRY node is used to define a location within a CSECT. It is used in a situation where reference to a Diana construct may occur from another compilation unit, but the reference is to a location other than the beginning of the CSECT. The referencing unit does not know the displacement within the CSECT, so the ENTRY node serves as a place holder. It is attached to the Diana construct represented by the ENTRY node. An ENTRY node has the following components:

1. the VMM locator of the CSECT node to which it is attached;
2. the displacement (in SUs) within the CSECT;
3. a function (same meaning as the function component of a CSECT);
4. a VMM locator for the defining Diana construct.

It should be noted that the CSECT and ENTRY nodes do not have names as do the equivalent constructs in a traditional object module. In such object modules, the names are bound by a linker (frequently known as a "binder"). Name binding would be superfluous in the MAPSE linker, because the binding has been performed by the compiler (in the semantic analysis phase) and preserved in the program library.

A CSUD (compiler SU definition) node is used to define the compile-time values of a contiguous array of storage units within the CSECT to which it is attached. (This corresponds to the .TXT record of a 370 object module.) It has the following components:

1. a displacement (in SUs) within the CSECT;

2. an array of SUs containing the machine code and/or data.

The compiler may define the storage units of a CSECT with one or more CSUD nodes. Some or all of the storage units of a CSECT may be left undefined.

An LSUD (linker SU definition) node is used to define the link-time values of storage units in the CSECT to which it is attached. (This corresponds to the .RLD record of a 370 object module, which defines relocation data.) The value is specified with an arbitrary expression containing a limited set of operators and operands as described below. An LSUD node contains the following components:

1. a displacement (in SUs) within the CSECT of the link time value;

2. a size (in SUs) of the link time value (right adjusted in the field);

3. a locator of a link-time expression node defining how the linker is to compute the value.

The link time value is typically 1 to 4 storage units in size, and typically contains an address. However, the linker will also assign values representing Ada exception identities, stack frame sizes, and other entities which the compiler cannot reasonably compute (due to separate compilation).

The link-time expression nodes, LOPR, LADR, LLIT, LATT, are used to represent the computation of a link-time value. The value is referenced by an LSUD node. A value of arbitrary complexity may be represented by a tree structure with LOPR branch nodes and LADR, LLIT, and LATT leaf nodes. A simple value may be represented with a single leaf node. Thus complex address constants or other link-time values may be defined by the compiler and computed by the linker.

An LOPR node represents an operation. It has the following components:

1. a binary operator (plus,minus,times,divide);

2. a VMM locator for the left operand;

3. a VMM locator for the right operand.

An LADR node represents the address of an entity. It has the following components:

1. a function (used to select from a set of CSECT or ENTRY nodes);

2. a VMM locator of a node (Diana construct, CSECT, or ENTRY node.)

The function component is needed only when the locator refers to a Diana node with multiple CSECT (or ENTRY) nodes. With separate compilation of bodies, the CSECT representing a procedure may not exist when a call to the procedure is compiled. The compiler creates a LADR node referring to the Diana construct for the specification. The linker can find the Diana construct for the corresponding body, and select the correct CSECT from the set of attached CSECTs by comparing the function components. If the body and the call are in the same compilation unit, the compiler may point the LADR node directly at the desired CSECT.

An LLIT node represents a literal. It has the following component:

1. a literal value of an implementation-defined integer type.

An LATT node represents an "attribute" (The word "attribute" is quoted here to distinguish its use as a Diana term, rather than a database term.) It has the following components.

1. an "attribute" selector (frame size, exception identity, etc.);

2. the locator of the node whose "attribute" is being referenced (typically a Diana construct).

An interesting example of the power of link-time expressions is the compiler's creation of a byte pointer for special byte instructions on a 16-bit word-addressed machine. The word address containing the byte is represented by a LADR node; a LOPR node (times) points to the LADR node and to a LLIT node (value 2), instructing the linker to double the word address; another LOPR node (plus) points to the LOPR (times) node and to a LLIT node (value 0 or 1 for the left or right byte as appropriate. A traditional linker with only limited operations on addresses (usually addition of a signed constant) could not create such a byte pointer.

The nodes described above are contained in the CONTENTS=>DIANA object produced by the compiler. These nodes are referenced by Diana construct nodes as code generator "attributes". In addition, the root node for each object contains a list of all CSECT nodes, a list of all LADR nodes (for unreferenced CSECT elimination), and a list of all Ada exception definitions. This enables the linker to quickly locate the nodes that are significant to its processing, without a complete walk of the Diana tree.

(e) Run Time Library. The Ada compiler generates code that calls run time routines for certain language constructs (allocators, tasking, exception raising, etc). These routines must be available to the linker, and there must be a convention for the compiler to use for the

run time routine CSECT references. It is desirable to code as much of these routines in Ada as possible, yet not allow direct reference to them by Ada source code. (For example, if an allocator "NEW t" were translated by the code generator into a call on a function "alloc_storage(t'size)" defined in the run time library, the Ada specification must be defined in such a way as to prevent a programmer from calling "alloc_storage" directly.)

This problem is resolved by placing the specifications of the run time library in the private part of a predefined package specification (for example package STANDARD.SYSTEM). Thus only the compiler itself will have visibility to the run time routines. This solution adds no additional complexity to the linker, since the reference mechanisms are the same as to user-compiled Ada programs.

3.2.3.2 Processing

Linker processing occurs in the following stages:

1. unit recompilation;
2. preamble and missing body generation;
3. special CSECT creation;
4. unreferenced CSECT elimination;
5. exception identity assignment;
6. CSECT placement;
7. memory image creation.

(a) Unit recompilation. This phase is similar to the unit recompilation processing in the compiler. A referenced unit list is constructed using the unit selection algorithm described in the compiler processing section. It differs in two ways. The process starts with a single compiled unit, rather than with source. The main unit as specified (or computed) is conditionally recompiled, resulting in a consistent set of referenced units. Unit recompilation proceeds as in the compiler. However, with the compiler's unit recompilation algorithm, only library unit specifications would be recompiled. The bodies corresponding to those specifications, and the subunits of those bodies, would not have been recompiled, since the references from the specification to the bodies are only implied (barring INLINE and GENERIC). Thus the linker's unit recompilation algorithm, unlike that of the compiler, selects the corresponding bodies and subunits, applying the same version selection algorithm. (Any units subject to recompilation which are identified with VERSION=>"*TEMPORARY" are not recompiled, so the preamble and body generation described in the next stage will occur for each link.) The linker program performs the unit recompilation by invoking the RECOMPILE program via the KAPSE program invocation interface. The parameters which the linker supplies to RECOMPILE are described in Section 3.2.4.1.

The main unit, from which the unit recompilation process starts, does not have to be the ultimate main unit. It may in fact be any library unit; the resulting link serves as a unit test link. This allows testing of individual compilation units without involving any of the higher level units. If the MAIN=> parameter is omitted, the linker will attempt to compute a main unit by the following method, which is based on the fact that the main unit will not be referenced by other units. The partition PER=>UNIT,USAGE=>NEW is opened; if there are multiple units in this partition, any units which have revisions or versions with USAGE=>(not NEW) are ignored. If there is only one unit left, it is chosen as the main unit. (Alternative main units may exist in a program library, in which case the desired main unit must be specified to the linker.) This frees the programmer from having to specify a main unit when there is only one reasonable candidate.

The main unit must be a library unit, either a package or a subprogram. If the subprogram has parameters, a preamble will be built during the next stage of processing.

(b) Preamble and Missing Body Generation. This stage involves the automatic construction of compilation units that have not been supplied by the programmer. If the main unit, as specified by the user or computed by the linker, is a function or a procedure with parameters, a preamble is built. The preamble generator is invoked, which analyzes the Diana for the main subprogram and generates source code for the preamble. The preamble is compiled using the source compiler, creating a unit which now acts as the main unit. (When the preamble executes, it converts the string values of parameters from the KAPSE, to the internal representations used in the parameters of the subprogram, and then invokes the subprogram.)

As a result of the unit processing stage, a list of missing bodies has been constructed, containing an entry corresponding to each specification for which a body has not been supplied by the user. The body generator (described in a later section) is invoked with this list of units. Like the preamble generator, it produces source text. This text is compiled using the source compiler, producing null executable bodies.

(c) Special CSECT Creation. The set of library units selected in the previous stages must be elaborated in the correct order when the program is executed. The elaboration order is computed using the dependency relations of library units. The order is recorded in an elaboration CSECT. This CSECT is a table of CSECT or ENTRY addresses, where each address refers to the elaboration function as created by the compiler for each library unit. The last address is of the main unit. The Ada runtime library contains a startup routine which uses the elaboration CSECT to sequence the library unit elaborations. The address of the startup routine is made the initial program counter or PSW for the Ada program context.

In addition to the elaboration CSECT, the linker creates a map CSECT which allows the run time system's debugging support routines to locate the statement hook tables of each compilation unit.

(d) Unreferenced CSECT Elimination. The use of packages and generics in Ada is likely to result in unreferenced subprograms. Since the compiler generates a CSECT for each subprogram body, an unreferenced subprogram results in an unreferenced CSECT. This processing stage identifies which CSECTs are not referenced by any LADR nodes and excludes them from placement in memory. Although this requires a pass over the LADR nodes prior to CSECT placement, it is worth the additional processing time to reduce the memory requirements of the final program.

(e) Exception Identity Assignment. Each exception defined in a linked unit is assigned a unique integer value to represent its identity in RAISE statements and exception handlers. A unique value is required so that the run time system's raise handler can locate the correct handler for an exception. A complicated scheme in the compiler to assign unique identities (across compilation units) is avoided, because the linker can simply assign consecutive integers to the set of all exceptions in the linked program.

(f) CSECT Placement. This processing stage assigns relative locations to all of the CSECTs not excluded by the previous stage. Pure and impure CSECTs are grouped in two separate segments. Each CSECT in turn is placed following the previous CSECT in the appropriate segment, with the alignment as specified in the CSECT node.

(g) Memory Image Creation. This processing stage allocates pure and impure memory image arrays (VMM variable length arrays) and defines the contents using the CSUD nodes of each CSECT. A storage unit "fill" value is defined for storage units not specified in CSUD nodes or skipped because of CSECT alignment. After all the CSUD nodes for a CSECT have been processed, the LSUD nodes are used to assign the link time values.

As a result of exception identity assignment and CSECT placement, all components of link-time expression nodes have associated values. Each LSUD node causes a link-time computation to be performed and a value stored into the pure or impure memory image at the location defined in the LSUD node.

3.2.3.3 Outputs.

There are two outputs from the linker; an executable program context, and a short link summary written to standard output. The program context is a composite object that is initialized by the linker to contain:

1. the pure and impure memory images;

2. a map object recording the selection and placement choices made by the linker;

3. a window on the program library which provides the debugger or any other program context analyzer with access to the program library which generated the program context.

The map object is a VMM file containing a mapping set for each of the following:

1. each linked compilation unit;

2. each placed CSECT;

3. each assigned exception.

The mapping sets are comprised of nodes whose membership testing criterion is the VMM locator of the corresponding linkable Diana construct, and with a component which defines the value assigned or selected by the linker. Given a program context, a program (for example the debugger) can use the map object to quickly find the value (unit version, CSECT address, exception number) selected or assigned by the linker.

3.2.3.4 Adaptation

(a) Placement Control. Linker commands to control placement of CSECTs in memory may be required for some target machines (other than the 370 and 8/32) which have non-uniform memory spaces. The non-uniformity may be due to differences in the addressing techniques, power consumption, speed, etc, and so may require user control over the otherwise automatic placement of CSECTs.

The linker commands would be read by the linker from its standard input, and consist of simple free-form syntax: command verbs followed by operands. Commands controlling placement fall into two categories:

1. segment definition commands which define areas of memory and the usage for the linker (pure, impure, reserved, etc.);

2. placement commands which direct the placement of CSECTs, either individually or by groups defined by compilation unit, function, etc.

(b) Overlay. Linker support of program overlay may be needed for target machines with limited memory. When an overlay requirement is established, the linker can be extended to process commands which define overlay regions and their contents.

(c) Relinking. The realities of target testing, validation, or release control often require that locations of CSECTs change as little as possible from link to link. To support this requirement, the linker can be extended to process commands which refer to a previous link. The map from the previous link is read and an attempt is made to place CSECTs at the same location as in the previous link. If a CSECT increases in size, the linker will place it in an unassigned location.

CSECT size padding commands may be added so that initial links may leave room for the growth of CSECTs and reduce the likelihood of changed CSECT locations. This can reduce the cost of revalidation.

3.2.4 Auxilliary Program Library Functions

3.2.4.1 RECOMPILE

This program allows a user to request recompilation of one or more units independently of the submission of source text to the compiler, or the invocation of the LINK program. This may be desirable in the following situations:

1. the entire program library (or a subset) is to be recompiled, using a different compiler release or different compiler options;

2. the units affected by a change in a referenced unit are to be recompiled explicitly, rather than automatically at link time.

The RECOMPILE program creates a revision of a compilation unit, using the same representation of the source as was submitted to the compiler. The algorithms used for unit recompilation are described in the source compiler and linker processing sections above. (The linker actually invokes this program to perform its recompilations.)

The user invokes the RECOMPILE program with the following parameters:

```
RECOMPILE UNITS=>unit_spec LIB=>prog_lib [options] --> summary
```

The units to be recompiled are specified by a list of names, or ALL. A subunit is specified by giving the library unit name and subunit name, as in "uname.sname". The body of a library unit is specified with "uname.body". The options that may be specified include all of the compiler's options except VERSION and REORDER (USE_VERSIONS,LIST,DEBUG,OPTIMIZE). An option specified via RECOMPILE overrides the corresponding option supplied with the source compilation of the given version. Options specific to the RECOMPILE program are:

```
BODY=>(YES,NO)
          -- YES means that a body corresponding to a
          -- specification is automatically recompiled if the
          -- specification has been recompiled since the last
          -- compilation of the body

          -- NO means that a body is not automatically recompiled
          -- given recompilation of its specification
          -- (it is recompiled if requested explicitly
          -- via the UNITS parameter)
```

```
FORCE=>(YES,NO)
          -- YES means recompile the specified units unconditonally
          -- NO means recompile only if necessary to get consistent
          -- usage of units
```

The linker invokes this program with the following options:

```
RECOMPILE UNITS=>main unit,
          BODY=>YES,
          FORCE=>NO,
          LIB=>same as on link
          USE_VERSIONS=>same as on link
```

3.2.4.2 Body Generator

This program creates the source form for a null body corresponding to a given specification. The specification may be for a library unit or a subunit stub. The program locates the specification in the program library. The processing which follows depends on the kind of unit being generated.

In the case of a subprogram, the Diana form of the specification is converted to source text form. If the subprogram is a subunit, the specification is preceded by the appropriate SEPARATE statement. A null body (BEGIN NULL; END;) is generated following the specification. In the case of a function or a procedure with OUT parameters, the value computed by the null body is undefined, as per [G-1, 6.5.]

In the case of a package, a package body skeleton is generated, preceded by a SEPARATE statement if a subunit. The Diana for the package specification is scanned for subprogram and package specifications. For each subprogram specification found, a subprogram body is generated as above. For each package, the routine that is handling the package is recursively invoked. Following the declarations of nested subprogram and package bodies, a null body for the outer package is generated.

In the case of a task body, the null body is generated without accept statements.

The body generator is invoked as follows:

```
BODYGEN UNITS=>unit_spec LIB=>prog_lib --> source text
```

The source text is written to standard output.

3.2.4.3 Preamble generator

The preamble generator is invoked by the linker or by the user explicitly. This program creates the source for a preamble. The preamble itself acts as a driver program for main programs which are written as functions or procedures with parameters. The preamble generator locates the specification of a main library unit subprogram P in the program library. The identity of P is supplied by the

invoker. A new procedure is generated in source form with the name DRIVER_P. The source of DRIVER_P:

1. has a WITH P statement which names the original main unit, as well as all of the units named in its WITH statements (so that subprogram parameter types are visible to the preamble);

2. declares a local variable for each parameter;

3. calls the KAPSE to get the string value for the actual parameters from the PARAMETERS attribute of the program context;

4. for each IN or INOUT parameter, uses the KAPSE function PICK_PARAM to locate either a named parameter-value pair, or the correct positional argument;

5. if a value is found it is converted to the internal representation for the type using the VALUE attribute of Ada; the value is stored in the corresponding local variable;

6. if the value is not found, a default value is assigned to the local variable; (the normal default mechanism of the compiler cannot be used, since it is not known whether the user will supply a value or not);

7. the main program P is called, passing the local variables as arguments;

8. the inverse conversion is performed (using the IMAGE attribute of Ada) for INOUT or OUT parameters or function value;

9. the KAPSE is called to record the results of the inverse conversions in the PARAMETERS attribute of the program context.

For each value found in (3) above, if the value is "?" or ":", the preamble writes to standard output the type name expected, along with the enumeration literals if the type is an enumeration, or a range if the type is numeric. This gives a default help capability. If any value is specified in this way, the main program P is not called; instead, a message is written to standard output requesting re-invocation of the program.

Automatic use of the preamble generator occurs if a link is performed with a main unit that has parameters. The preamble generator may also be used explicitly, with the user saving the generated source, perhaps modifying it with more detailed help output. Thus a standard user interface is insured for all user-written programs. However, in case of special requirements the normal conventions may be purposefully overridden by modifying the preamble source.

The preamble generator writes the source for the preamble to standard output. It is invoked as follows:

```
PREAMBLE UNIT=>name LIB=>prog_lib --> source text
```

3.2.4.4 Change Analyzer

This program takes two versions or revisions of a unit (U and U') and compares the Diana trees. The Diana tree of U is walked in a recursive algorithm which visits each construct. Two techniques for comparision are used:

1. if the Diana construct in U is the defining occurrence of an identifier, the same identifier is looked up by name in the corresponding scope of U',

2. otherwise, a structural tree comparison is performed, stopping at the first unequal comparision.

The output of the comparison is a VMM subdomain that contains a mapping set. Each element of the set is a comparision node, with its membership testing criterion being the VMM locator of the node in U. The comparison node contains the VMM locator of the corresponding node in U'. If the comparison node represents the defining occurrence of an identifier, the comparison node records the result of the comparison (equal, not equal, not found). Nodes compared as a result of structural tree comparison are in the set only if they compared equal. Nodes are considered equal if all semantic and storage allocation "attributes" are equal.

After the mapping set is built, an inverse set is constructed with the membership criterion being the VMM locator of the node in U'. Then a structural walk of the Diana tree of U' is performed, checking that each node representing a defining occurrence of an identifier is in the inverse set. If not, a new member of the set is created indicating that the corresponding node in U is not found. (Thus adding a new identifier in U' is properly represented in the inverse set.)

The VMM subdomain produced has the two mapping sets, a flag indicating whether all comparisons succeeded, and a flag indicating whether every pair of VMM locators in U and U' have identical subdomain offsets. The last flag allows the MAP program to optimize the mapping by instructing the VMM system to substitute only the subdomain number part of a VMM locator.

The program optionally produces a human-readable output showing the differences found by the comparison.

The mapping sets are used by the MAP program to determine which units need recompilation.

3.2.4.5 MAP: The Domino Stopper

The MAP program gets its nickname from its ability to stop the "domino effect" of unit recompilation, described in the special requirements section of the compiler specification (3.2.2.4). The user invokes this program with two versions or revisions of a

compilation unit. By so doing, the user is asserting that the two are identical, or at least similar enough so that some or all of the referencing units do not need recompilation.

The change analyzer program is invoked to produce the mapping sets between the two units, U and U'. Each unit in the library is inspected to see if it refers to unit U. If it does, the global cross reference set in the unit is inspected for individual references to identifiers in unit U. For each identifier used, the mapping set produced by the change analyzer program is checked to see if the identifier is identical in U'. If all identifiers used in U' are found to be identical U', the unit does not need recompilation. Note that if U is a package and the referencing unit has a USES statement, U and U' must be identical for all identifiers, not just the referenced ones.

If any used identifiers are found to be changed or deleted in U', the unit must be recompiled. It is recompiled immediately, to avoid conflict with the mapping set to replace U.

After all referencing units have been checked, if any have not been recompiled, the mapping sets built by the change analyzer are processed to produce the VMM locator association set from U to U', which is installed as a replacement for U. The VMM system will then map any references to nodes in U with the identical nodes in U'.

The MAP program is invoked with as follows:

```
MAP FROM=>vmsd_num,TO=>vmsd_num,LIB=>prog_lib --> summary
```

Each vmsd_num is the value of the VMSD attribute of the versions/revisions which are asserted to be the same.

3.2.4.6 Object Module Converter

The Ada language makes provision for the incorporation of non-Ada programs into the program library with the PRAGMA INTERFACE. The object module converter program converts the output of a foreign translator (compiler or assembler) into a compilation unit of the program library. The output of the translator is in a particular format called an object module. The object module format is as expected by the linker for the non-APSE system. This is usually a common format for all translators for a given machine-operating system combination. This program installs the object module in the program library by:

1. finding the corresponding Ada specification for the subprogram, which was identified by PRAGMA INTERFACE;

2. creating a new PER=>UNIT object representing the body of the specification;

3. defining the code generator attributes of the body by reading the object module (usually a sequential operation) and constructing CSECT, ENTRY, CSUD, LSUD, and link-time expression nodes as required.

The compiler supports PRAGMA INTERFACE only for subprograms which would otherwise have separately compiled bodies. (Without this restriction, the CONTENTS=>DIANA object produced by the compiler and containing an INTERFACE subprogram would have to be modified by the object module converter.) Thus, PRAGMA INTERFACE may be given only for library unit subprograms or subprograms which satisfy the language requirement for subunits. A certain level of support for a translator's run time system is needed to allow its object module to properly execute within the Ada run time system. Thus restrictions on some foreign language features (such as I/O or supervisor-system calls) may apply to a given combination of foreign translator, object module converter, and host system.

3.2.4.7 Program Library Status Report Processor

This program produces a human readable listing which summarizes the current state of a program library. For each version of a unit, the listing reports:

1. the unit identity: name, version, part, kind, VMSD number;
2. when it was compiled;
3. the options it was compiled with;
4. which units it referenced (by name, version, and VMSD number);
5. whether it requires recompilation;
6. which units depend on it (by name, version, and VMSD number).

The program has an option to invoke the link map lister (described below) for each PER=>LINK object in the library. An option to delete unused or out of date PER=>UNIT and PER=>COMPILATION objects may also be supplied.

3.2.4.8 Link Map Lister This program reads the map object produced by the linker and creates a readable listing of its contents. The listing shows:

1. each unit in the link (name, version, part, kind, VMSD number);
2. each control section (CSECT): its location and size;
3. each exception: its scope and unique number.

3.2.4.9 Source Extractor

This program extracts a text file from a program library for a compilation unit. The compressed source representation in the PER=>COMPILATION object is expanded and written as source text lines to standard output. Source text that was submitted with the compilation unit representing other compilation units or the contents of PRAGMA INCLUDE files is not written. The program is invoked as follows:

EXTRACT UNIT=>name LIB=>prog_lib -->source text

3.2.4.10 Program Library Creator

This program is invoked by the compiler when the named program library does not exist. It creates a program library composite object by copying the standard system library, which contains the pre-defined library units STANDARD, CALENDAR, SHARED_VARIABLE_UPDATE, UNCHECKED_DEALLOCATION, UNCHECKED_CONVERSION, INPUT_OUTPUT, TEXT_IO, and the KAPSE packages described in the KAPSE database specification document [I-3].

4.0 QUALITY ASSURANCE

Testing of Program Integration modules will proceed in three stages:

1. Unit testing in the "bootstrap" environment, to verify algorithms used to implement the individual program modules.

2. Integration testing:

   a) Access system testing with test driver modules. This will test the program library access packages as a general purpose capability.

   b) Tool integration testing in the final host environment.

3. Acceptance testing using a set of user scenarios appropriate to Ada Programming Support tool and Embedded Computer System applications development.

5.0 NOT APPLICABLE

6.0 NOTES

Ada supports separate compilations and separate compilations of specifications and bodies, at that. A complete embedded computer system (ECS) software product might involve hundreds of interdependent compilation units and thousands of compilations and re-compilations. It is imperative that these be conducted in an efficient manner. The potential problem can be illustrated by considering the 26 compilation units, A through Z, where initially the units are compiled as follows:

```
A with B

B with C

C with D

.
.
.
Y with Z
```

In the general case, A through Y depend upon Z, A through X depend upon Y, ... A through C depend upon D, etc. The indirect dependence of A on C through Z must be assumed in the absence of detailed analysis of each unit. (The MAP program performs this analysis.)

1. Suppose the logic of compilation were followed whereby whenever a unit is compiled, all units dependent on it are re-compiled. Thus, if the compiler were to receive (Z, M) together in that order, Z would be compiled plus Y through A, totaling 26 compilations. Then M would be re-compiled (it just was compiled) plus L through A, again, adding another 13 compilations, totaling 39. Note that if Z were to be compiled again, perhaps to add a statement or make another modification, another 26 re-compiles would occur even though the program had never been linked for execution.

2. Suppose a different approach were followed, one which compiled just the units presented, followed by all necessary re-compilations at the end to ensure a consistent set. Thus with (Z, M), Z is compiled, M is compiled using the old N through Z, and then Y through A are recompiled. This is a total of 27 compilations, a significant improvement.

3. Both of the above methods, however, suffer from burden of preparing a consistent set of units without any knowledge of whether this set will be used; i.e., linked or otherwise examined within the program library. The current design compromises here in favor of the contention that delaying consistency until consistency is really required is the most efficient course. The approach (subject to user over-ride) is a follows: partially compile all presented units (i.e., through lexical and syntactical phases) and complete the compilation simply; for (Z, M) this means compile Z and then notice that M depends on Z; i.e., the new Z, so recompile Y

through N with the new Z and then compile M. This totals 14 compilations. Of course A through L still refers to old units and are thus inconsistent but this will be reconciled when the program is used (linked). In the mean time, Z and M could be compiled again causing another 14 recompiles instead of 27 or 39 for the rejected approaches.

To reduce recompiles even further, the user can assert that a new Z doesn't affect the other units (because only a comment was changed, for example). The MAP tool checks the assertion and, if true, no further recompilations will occur because of the new Z.

Summary			MAPSE Design
	(1)	(2)	(3)
Recompiles at Compile Time	39	27	14
Recompiles at Link Time	0	0	12

The recompile strategy of the design will significantly reduce the potential for extraneous compilations during program development. One other point here about the desgin. With hundreds of compilation unit's the correct order of compilations can become near impossible for the user to sort our manually. The current MAPSE design provides an automatic assist in this task by providing an ordering capability in the compiler. After partial compilation, the tool examines the units and properly orders them for continued compilation. Thus, in effect, the user can submit compilation units in any order, and be assured of a correct compilation, if the source contains no errors.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.


DATE
ILME